Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MATTHIAS SCHÖPE
MASTER THESIS

QDACITY QUALITY METRICS

Submitted on August 22, 2017

Supervisors: Andreas Kaufmann, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, August 22, 2017

License

Erlangen, August 22, 2017

Abstract

The cloud-based QDAcity platform allows to conduct collaborative research projects applying Qualitative Data Analysis methods and research validation through crowdsourcing. When using Qualitative Data Analysis as a research method, expressive measurements of the quality and maturity of the results are essential to prove the validity of the research findings. Common measures for validity of ratings are inter-coder agreement metrics, and for measuring the maturity of a qualitative research project, a prevalent approach is to calculate saturation. However, with a high number of raters, inter-coder agreement metrics become inconvenient to evaluate and the calculation of saturation requires a clean documentation of many project variables. The cloud environment of QDAcity can solve both of these problems, because it can effciently store all ratings and project variables, and thus integrate both metrics more conveniently for the researcher. This thesis presents the implementation of the two inter-coder agreement metrics Krippendorff's Alpha and Fleiss' Kappa in QDAcity and a new approach with its implementation of theoretical saturation.

Contents

1 Introduction 1
1.1 QDAcity 1
1.2 QDAcity Use Cases 2

2 Problem Definition 3
2.1 Inter-coder Agreement Metrics 3
2.2 Saturation Metrics 4
2.3 Requirements 4

3 Technologies 6
3.1 Google App Engine 6
3.1.1 Architecture of GAE applications 6
3.1.2 DataStore for persistence 7
3.1.3 Memcache 7
3.1.4 TaskQueues for scalability 8

4 QDAcity 9
4.1 Projects 9
4.2 Code Systems 10
4.3 Textdocuments 11
4.4 Validation Reports 12

5 Quality Metrics 13
5.1 Related Work 13
5.2 Approach for Implementing Inter-coder Agreements Statistics in QDAcity 14
5.2.1 General Process 14
5.2.2 Report Data Structures 16
5.2.3 Extraction of Input Data for Quality Metrics 17
5.3 Krippendorff's Alpha Coefficient 18
5.3.1 Interpretation of Krippendorff's Alpha results 19
5.3.2 Implementation Details 19

5.3.3 Discussion of Krippendorff's Alpha implementation 21
5.4 Fleiss' Kappa 22
5.4.1 Interpretation of Fleiss' Kappa 23
5.4.2 Implementation Details 24
5.4.3 Discussion of Fleiss' Kappa implementation 24

6 Saturation **25**
6.1 Related Work 25
6.2 Definition of Saturation in QDAcity 25
6.3 Technical approach 27
6.3.1 Logging of Changes 27
6.3.2 Calculation of Saturation 28
6.3.3 Saturation Parameterization 30
6.3.4 Implementation Details 32
6.3.5 Discussion of saturation feature 36

7 Evaluation **38**
7.1 Requirements 38
7.2 Correctness of Quality Metrics 39
7.2.1 JUnit Tests for Krippendorff's Alpha 39
7.2.2 JUnit Test for Fleiss' Kappa 40
7.3 User's benefit 40

8 Conclusion **41**
8.1 Future Work 41

Glossary **42**

Acronyms **43**

References **44**

List of Figures

3.1 Basic architecture of Web Applications on Google App Engine . . 7

4.1 Simplified UML Class Diagram of Project and related classes in QDAcity 9
4.2 Example for a Coding System from a real QDAcity project 10
4.3 Simplified UML Class Diagram of Code class in QDAcity 11
4.4 Example for a coded paragraph in a textdocument and its HTML representation 11

5.1 General process of inter-coder agreement report generation in QDAcity 14
5.2 Simplified UML Class Diagram of ValidationReport and ValidationResult 16
5.3 Textdocuments - Units and their codings by different raters 17
5.4 Simplified UML Class Diagram of AlgorithmInputGenerator and classes 20
5.5 Simplified UML Class Diagram of KrippendorffsAlphaCoefficient and related classes 20

6.1 Outline of general working process in a QDAcity project 26
6.2 Simplified UML class diagram of SaturationCalculator and related classes 32
6.3 Process of calculating a new SaturationResult in QDAcity 33
6.4 Comparison of latency of update code calls with deferred and direct change logging on F1 instance 35
6.5 Comparison of cumulative latency of update code calls with deferred and direct change logging on F1 instance 35
6.6 Comparison of latency of update code calls with deferred and direct change logging on F4 instance 35
6.7 Comparison of cumulative latency of update code calls with deferred and direct change logging on F4 instance 36

List of Tables

1.1 Overview of typical QDAcity use cases 2

3.1 Push Queues and Pull Queues in Google App Engine 8
3.2 Push Queues and Pull Queues in Google App Engine 8

5.1 Popular inter-coder agreement metrics and their characteristics . . 13
5.2 Reliability Data as input for Krippendorff's Alpha 18
5.3 Coincidence Matrix for Krippendorff's Alpha 18
5.4 Interpretaion of Krippendorff's Alpha results 19
5.5 General form of input data for Fleiss' Kappa 23
5.6 Interpretaion of Fleiss' Kappa results 23

6.1 Forms of saturation 26
6.2 Tracked Change Objects and their possible Change Types in QDAcity 29
6.3 Default weighting of change categories in QDAcity 30
6.4 Default maximum saturations of change categories in QDAcity . . 31

1 Introduction

With the wide distribution of cloud computing and the emerging of related technologies software applications can be taken to a new scale. Especially for collaboration within teams, a cloud application has several advantages, such as failure safety, enhanced global availability and scalability along with the number of users.

QDAcity is a Qualitative Data Analysis (QDA) tool which is accessible through a web browser[1] running on Google App Engine (GAE).

1.1 QDAcity

QDAcity is a cloud-based software tool for computer-assisted qualitative data analysis. It provides an environment for researches conducting qualitative research and supports the theory building process. This includes commonly known concepts from qualitative research such as a code system consisting of the researcher-generated codes. These are used to attribute data and to provide an interpretation or meaning of this data for the theory developement (Saldaña, 2015, p.4f).

A QDAcity project contains textdocuments which need to be analyzed. For analysis of these textdocuments, a code system is used, consisting of the relevant codes of the project's domain. The code system can be defined and redefined by the users of the project. A number of users act as raters, which means that they individually apply the codes to specific text segments of the documents in a project. This process is called *coding*. In the end of a project, users can write a theory about the coded textdocuments, which is the result of the theory building research.

[1]https://www.qdacity.com/

Table 1.1: Overview of typical QDAcity use cases

Name	Use Case	Project Leaders	Raters	Code System
UC1	University courses	1 (usually course leader)	around 10 to 300 students	**A:** only defined by raters **B:** Predefined by project leaders **C:** Students only write theory about coded data
UC2	Research groups	1 or 2	2-3	defined by project leaders, optimized by raters
UC3	Students writing thesis	1 (student writing a thesis)	usually 1	defined and optimized by project leader

1.2 QDAcity Use Cases

There are three typical use cases for QDAcity as shown in Table 1.1. The first use case (UC1) comprises university courses about qualitative research methods using exercises with defined educational objectives. In this use case, there is usually one person responsible for the project, and a number of students code the documents of the project. Depending on the course's educational objectives the students who act as raters can either define a code system by themselves (UC1 A) or get a predefined code system (UC1 B) or even get complete coded data only for writing a theory about it (UC1 C).

The second use case as shown in Table 1.1 (UC2) comprises of research groups who are in need of a qualitative data anaysis tool and were using competing products like MaxQDA[2]. In research groups, there are typically one or two people responsible for a project. Because the coding process requires a lot of time and domain knowledge by the raters, there are usually 2-3 raters in a research group.

The third use case (UC3) as presented in Table 1.1 is a student from social sciences or educational science who needs a qualitative data analysis tool for his or her thesis. In this case, one person is responsible for the project and also codes the data. In UC3, the student, who writes the thesis, defines and optimizes the code system.

[2]http://www.maxqda.com/

2 Problem Definition

In QDAcity, large number of raters can annotate the textdocuments of a project with codes. Over time new documents and codes for annotation are added to the project, or codes get changed and users complete their ratings by annotating the textdocuments, until the number of new discoveries through the gathering of additional data decreases. An essential question is how the quality of such a QDA project can be assessed in QDAcity.

This thesis focuses on two aspects of this question:

(a) How can inter-rater reliability methods be integrated as a validation method for quality of the analysis?

(b) How can the stopping criterion of the iterative process of data gathering and analysis be defined and used as an assessment of the progress and the quality of the project's state?

2.1 Inter-coder Agreement Metrics

Measuring the quality of the results of QDA in an objective manner is a difficult proposition, since the process of QDA (Miles & Huberman, 1994, p. 8f) is inherently interpretative and subjective (Kolbe & Burnett, 1991, p. 246f). The demonstration of the quality in qualitative research projects often rests solely in the rigorous following and documentation of codified methods. However, methods for an evaluative validation exist, such as comparing codings of the same data by different researchers through inter-coder agreement ((Krippendorff, 2004) and (Kolbe & Burnett, 1991, p. 249)). To this end, QDAcity currently supports the F-Measure ((Ricardo Baeza-Yates, 1999, p. 75f) and (Powers, 2011)).

A goal of this thesis is to enhance the usefulness of inter-coder agreement as a validation technique by adding an alpha coefficient and a kappa statistic as recommended statistics for inter-rater reliability ((Krippendorff, 2012, p. 267f) and (Viera & Garrett, 2005)).

2.2 Saturation Metrics

A QDAcity project can be considered as finished, when further activity would not have a significant effect on the project results. For the users of QDAcity, a metric providing an indicator of this case would help to estimate the proceeding of the project. When this state is reached, the project has reached saturation (Guest, Bunce, & Johnson, 2006). As QDAcity projects can be from various domains, different users have a different notion of progress made so far within a project. Therefore, a flexible metric with a useful default configuration for most use cases would provide a significant improvement for users to get information about their project progress.

Users are mostly interested in the historical developement of the saturation in their project in order to see the progress made in the project.

Currently QDAcity does not have a measure for the saturation of a project. In this thesis, a new approach for measuring the progress in a project is implemented.

2.3 Requirements

Derived from the above sections, we define the following requirements for the quality metrics and saturation feature for QDAcity:

1. The results of inter-coder agreement metrics shall contain the following presentations:

    1. a) A single value representing the average

    1. b) Agreement by code

    1. c) Agreement by rater

2. The architecture for inter-coder agreement metrics should be extensible and provide a generic environment so that any feasible inter-coder agreement metric can be implemented. Storage and representation of the agreement shall be abstracted from the used algortihms.

3. A saturation metric should provide an average single-valued result with configurable weighting and maxima. The result of a saturation metric should also allow to view the single values of the components of the average saturation.

4. The saturation feature shall show the historical developement of saturation.

5. The saturation metric shall be configurable in the following parameters:

the weight of a single component in the average saturation

the maximum needed by a single component to reach saturation

6. A reasonable preset for most use cases shall be provided for the saturation metric.
7. Vendor lock-in to GAE should be avoided.

3 Technologies

In this chapter, the technology stack with relevant technical details for the QDAcity project are presented.

3.1 Google App Engine

QDAcity is developed and hosted in the GAE cloud computing platform. GAE[1] is a platform-as-a-service for web-applications provided by Google Inc (Google Inc., 2008).

3.1.1 Architecture of GAE applications

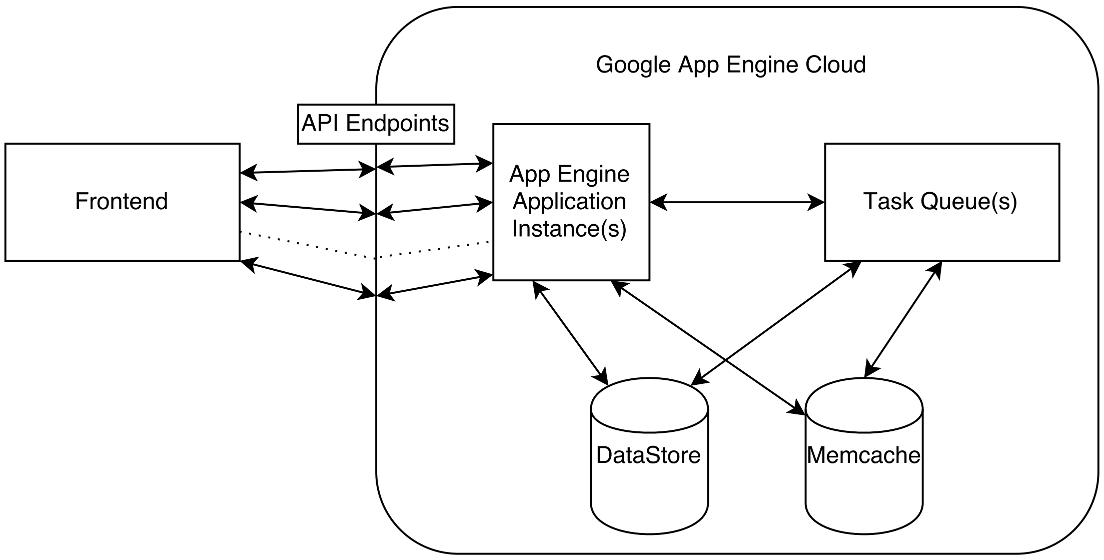
The basic architecture of GAE applications shown in Figure 3.1 is a client-server model oriented towards the well-known Model View Controller (MVC) (Erich Gamma & Vlissides, 1994) software architecture pattern.

The backend is written in Java 7 and provides Endpoints(Google Inc., 2017j) for communicating to the outside world through a Representational State Transfer (REST) interface using HyperText Transfer Protocol Secure (HTTPS) as communication protocol and JavaScript Object Notation (JSON) as data exchange format. The backend of a GAE application corresponds with the Controller in the MVC pattern. The backend includes the App Engine Application and the Task of the Task Queues as shown in Figure 3.1.

The frontend is written in JavaScript and Hypertext Markup Language (HTML) and communicates with the backend through the endpoints described above. The frontend is running on a client using a web-browser. The frontend of a GAE application corresponds to the View in the MVC pattern.

[1]https://cloud.google.com/appengine/

Figure 3.1: Basic architecture of Web Applications on Google App Engine



Based on: (Google Inc., 2017c)

3.1.2 DataStore for persistence

The DataStore(Google Inc., 2017d) as shown in Figure 3.1 of the GAE platform is a No Structured Query Language (NoSQL) document database. In the DataStore, GAE applications can store objects from their model, which corresponds with the Model of the MVC pattern. The model is defined through Java Data Objects (JDO), and a JDO implementation can be used to access the datastore as well as a low level datastore API (Google Inc., 2017e). QDAcity relies mostly on JDO but uses the low level API where appropriate to improve performance.

3.1.3 Memcache

The Memcache (Google Inc., 2017f) as shown in Figure 3.1 is a memory cache service which allows to temporarily store values or even complete data objects with a maximum to 1 MiB. The Memcache is significantly faster than the DataStore. However, values can expire from the Memcache any time, and therefore the Memcache cannot be used for persistence. There is no guarantee that a stored value in the Memcache can be read again out of the Memcache. The Memcache is intended to speed up operations where multiple accesses to the same objects in the DataStore are expected.

Table 3.1: Push Queues and Pull Queues in Google App Engine

Name	**Description**
Push Queue (Google Inc., 2017g)	Tasks can be defined and pushed into the queue. Push queues automatically consume tasks, scale and clean up.
Pull Queue (Google Inc., 2017i)	Allows to design the process of consuming app engine tasks. Scaling needs to be done manually, tasks explicitely deleted.

Table 3.2: Push Queues and Pull Queues in Google App Engine

Name	**Description**
Manual Scaling	Need to initialize explicitely, rely on memory state over time
Basic Scaling	Instance is created when request received. Turn down instanc on idle. Ideal for user driven activity.
Automatic Scaling	Request rate, response latence and other application metrics influence automatic scaling.

(adopted and abridged from (Google Inc., 2017a))

3.1.4 TaskQueues for scalability

Task Queues (Google Inc., 2017h) as shown in Figure 3.1 are used to asynchronously schedule work outside of user requests. Tasks are added to a Task Queue and executed later by scalable GAE worker services. There are two types of Task Queues, which are push queues and pull queues as outlined in Table 3.1. The Task Queues are configurable in the *queue.xml* file of the GAE project. In this thesis, only push queues (Google Inc., 2017g) are used.

A GAE instance can be used with three different scaling types as detailly explained in (Google Inc., 2017a). These are manual scaling, basic scaling and automatic scaling as outlined in Table 3.2. In this thesis, only auto scaling is used.

Push queues on an auto scaling instance impose a strict deadline of 10 minutes for each request (see (Google Inc., 2017g)). If a task becomes too big to be finished within 10 minutes, a strategy could be to split the task in smaller tasks and make use of the auto scaling feature, which starts new worker threads for parallel execution of tasks.

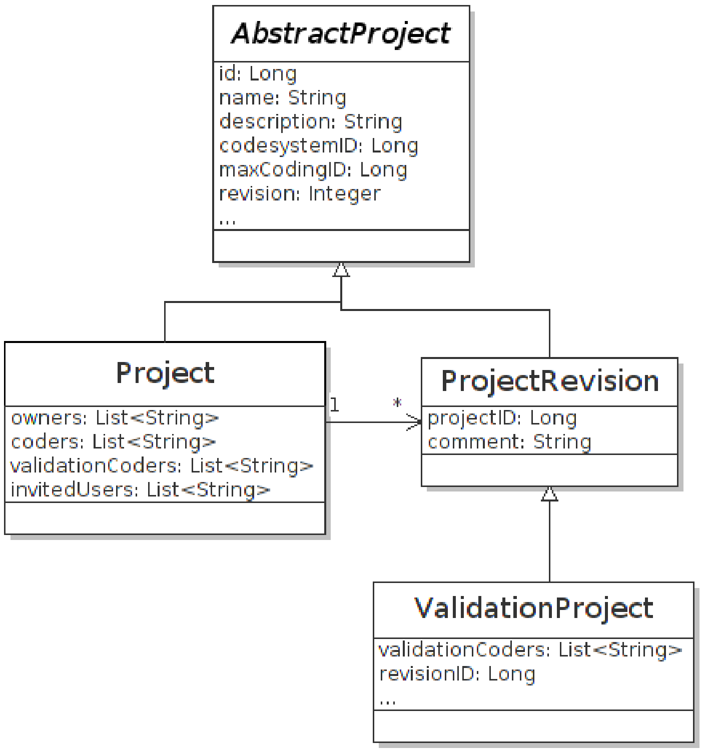
In case of failure, a task will be automatically restarted in a push queue. This behavior can be prevented, by setting the *task-retry-limit* of the queue in the *queue.xml* configuration file to *0*.

4 QDAcity

In this chapter, relevant aspects of QDAcity, its functions, features and implementation details are presented. In addition a rough overview of the relevant classes using Unified Modeling Language (UML) class diagrams is given. Following chapters are refering to this chapter, when describing implementation details.

4.1 Projects

Figure 4.1: Simplified UML Class Diagram of Project and related classes in QDAcity

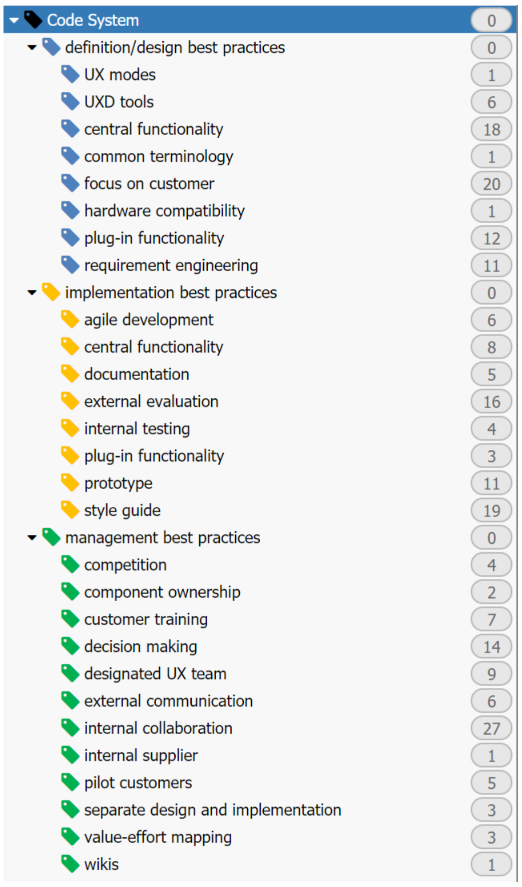


As outlined in Figure 4.1, the classes *Project* as well as *ProjectRevision* inherit from *AbstractProject*. Project revisions are intended to mark milestones in projects and develop them over time. Validation projects are necessary for the multiple raters of the textdocuments of the project respectively project revision. Raters have their own *ValidationProject* instance with their own copies of the textdocuments, so that they can only see their own codings.

A QDAcity project its revisions and validation projects contain textdocuments and a code system, which are presented in the next two sections.

4.2 Code Systems

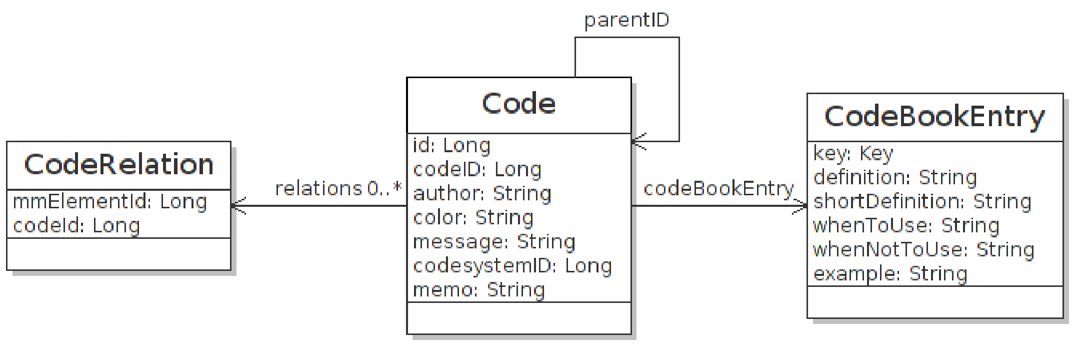
Figure 4.2: Example for a Coding System from a real QDAcity project



Code System	0
definition/design best practices	0
UX modes	1
UXD tools	6
central functionality	18
common terminology	1
focus on customer	20
hardware compatibility	1
plug-in functionality	12
requirement engineering	11
implementation best practices	0
agile development	6
central functionality	8
documentation	5
external evaluation	16
internal testing	4
plug-in functionality	3
prototype	11
style guide	19
management best practices	0
competition	4
component ownership	2
customer training	7
decision making	14
designated UX team	9
external communication	6
internal collaboration	27
internal supplier	1
pilot customers	5
separate design and implementation	3
value-effort mapping	3
wikis	1

The structure of a Code System is shown in the example in Figure 4.2. Code Systems in QDAcity are hierarchical and always have a root code named *Code System*. In Figure 4.2 two attributes of codes are visible - the code name and its color, indicated left of the code name. Written on the right side is the number

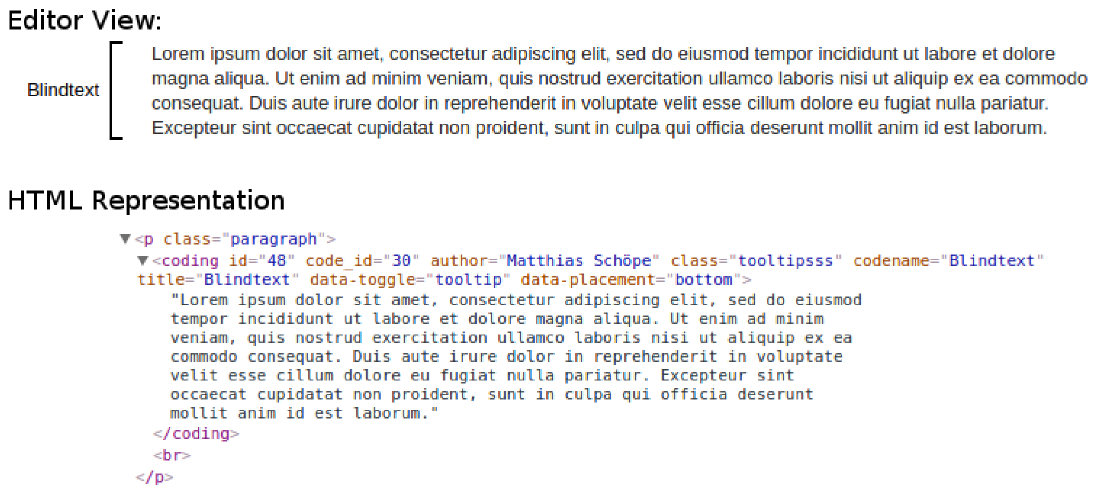
Figure 4.3: Simplified UML Class Diagram of Code class in QDAcity



of times a code has been applied. Further attributes of codes are outlined in the UML class diagram in Figure 4.3. Every code has a number of flat attributes, but also a reference to a *CodeBookEntry* with further attributes. Additionally a code can have several *relations* as shown in Figure 4.3 which allow code relations not only between parent and child codes, but with any code in the code system. A *Code* instance always has an *id* and a *codeID* as shown in Figure 4.3. The *id* is used for having a unique reference in the DataStore. The *codeId* is used as a unique identifier within the code system. Different *Code* instances may have the same value for *codeId*, but then they can not be part of the same code system. Unlike the other attributes *codeId* and *id* never get changed.

4.3 Textdocuments

Figure 4.4: Example for a coded paragraph in a textdocument and its HTML representation



Editor View:

Blindtext

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

HTML Representation

```
▼<p class="paragraph">
 ▼<coding id="48" code_id="30" author="Matthias Schöpe" class="tooltipsss" codename="Blindtext"
 title="Blindtext" data-toggle="tooltip" data-placement="bottom">
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
    veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
    velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
    occaecat cupidatat non proident, sunt in culpa qui officia deserunt
    mollit anim id est laborum."
  </coding>
  <br>
 </p>
```

A textdocument in QDAcity contains the text which users can code in their validation projects. A textdocument contains its text in HTML with additional self-defined tags. It can be split into textual units, such as paragraphs or sentences. In their validation projects users have a copy of the original textdocument, so that users only see their own codings and can individually code. The applied codes are directly saved into the textdocument's HTML structure. Therefore, the applied codes of a unit can be programatically accessed by parsing the textdocument using e.g. the JSOUP open-source Java library[1] and accessing the *coding* tag of the unit and look for the *code_id* attribute. An example is presented in Figure 4.4, where a coded paragraph and the corresponding HTML representation is shown.

4.4 Validation Reports

A validation report in QDAcity is a report triggered by a user on the frontend to calculate the agreement between coders based on the coded textdocuments of the project's revision. Due to the complexity of its calculation a validation report is processed in several tasks in a TaskQueue (subsection 3.1.4). After the report is finished its components are stored in the DataStore subsection 3.1.2, which are the validation report itself and the attached results. The user can then access the validation report through the dashboard of the project on the frontend. More details about validation reports and the process are explained in subsection 5.2.1 and subsection 5.2.2.

[1]JSOUP open-source Java library https://jsoup.org/

5 Quality Metrics

In this chapter, the approach and the implementation to question (a) of chapter 2 is presented and discussed. First there is an overview of related work and then details of the two implemented quality metrics in QDAcity are highlighted including relevant implementation details.

5.1 Related Work

The use of inter-coder agreement metrics is a commonly known approach in computational linguistics, specifically Natural Language Processing (NLP) (Artstein & Poesio, 2008, p. 1f), (Bruce & Wiebe, 1998) and (Joyce, 2013). As discussed in (Artstein & Poesio, 2008, p. 2f), several approaches exist for calculating an inter-coder agreement metric, such as κ (Kappa) Statistics e.g. (Cohen, 1960),(Fleiss, 1971) or others like Krippendorff's α (Alpha) in its original version from (Krippendorff, 1970) and later version (Krippendorff, 2007) refined in the context of computing.

The characteristics of popular inter-coder agreement metrics are compared in Table 5.1. An important property of a metric is the number of raters (#Raters

Table 5.1: Popular inter-coder agreement metrics and their characteristics

Name	**#Raters**	**Data**	**Rating type**
Krippendorff's Alpha (Krippendorff, 1970)	≥2	binary, nominal, ordinal, interval, ratio, polar, circular	categorical
Scott's pi (Scott, 1955)	2	binary or nominal	categorical
Cohen's Kappa (Cohen, 1960)	2	binary or nominal	categorical
Fleiss' Kappa (Fleiss, 1971)	≥2	binary or nominal	categorical

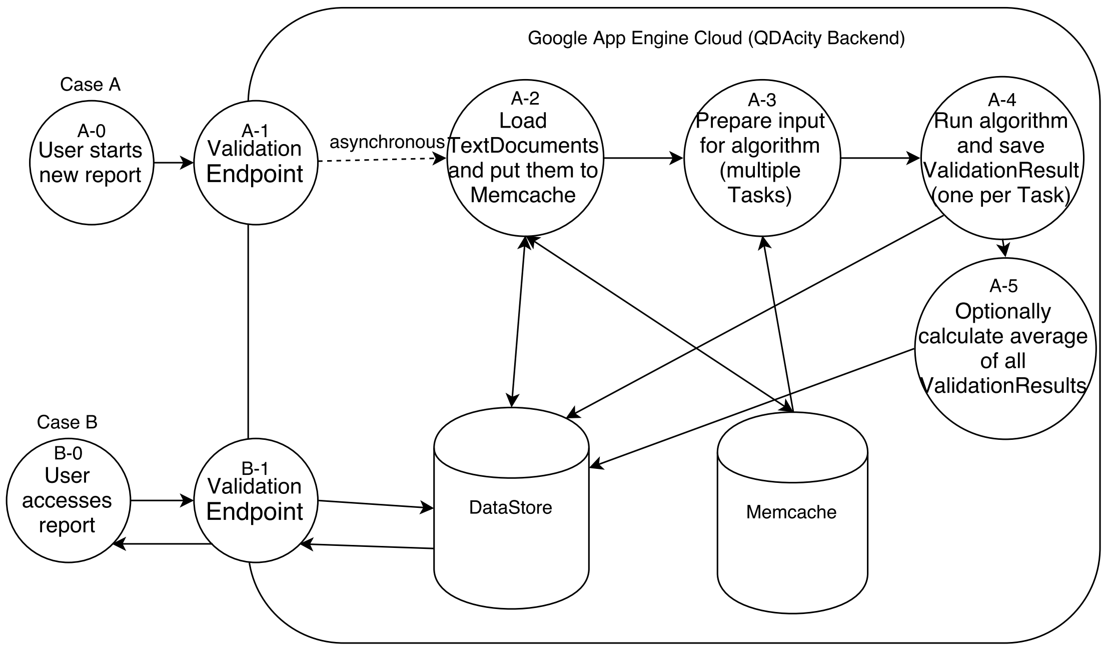
in Table 5.1) the metric can be applied on. It is important to note, that Scott's pi and Cohen's Kappa are only applicable between two raters. Furthermore all the presented metrics imply that the coders use a categorical rating, which means, that no unit can be coded with different categories at the same time by the same rater as they are mutually exclusive.

5.2 Approach for Implementing Inter-coder Agreements Statistics in QDAcity

This section focuses on the general architecture of the framework around the inter-coder agreement metrics, which I implemented in this thesis. The general process and its generic reusable components as well as the persisted data structures are discussed in detail.

5.2.1 General Process

Figure 5.1: General process of inter-coder agreement report generation in QDAcity


Google App Engine Cloud (QDAcity Backend)
Case A
A-0 User starts new report
A-1 Validation Endpoint
asynchronous
A-2 Load TextDocuments and put them to Memcache
A-3 Prepare input for algorithm (multiple Tasks)
A-4 Run algorithm and save ValidationResult (one per Task)
A-5 Optionally calculate average of all ValidationResults
Case B
B-0 User accesses report
B-1 Validation Endpoint
DataStore
Memcache

(See also Figure 3.1 for basic GAE architecture)

As discussed in the previous section, several metrics for implementation exist. The approach implemented in this thesis allows to use different metrics by defining a generic process in QDAcity for calculating any feasible inter-coder agreement metric as illustrated in Figure 5.1. Case A outlines the creation of a new report triggered by the user. Case B shows the process of viewing a persisted report by the user.

In step A-0 from Figure 5.1 the user selects the textdocuments to be analyzed, chooses the inter-coder agreement metric and provides a name for the report. The user clicks a start button and the users request is send via HTTPS to the ValidationEndpoint of the QDAcity instance running on the GAE. The ValidationEndpoint handles the users request (step A-1) and starts a new Task in a TaskQueue (see subsection 3.1.4). The ValidationEndpoint replies to the user's request with a HyperText Transfer Protocol (HTTP) 204 "No Content" [1].
The user can continue to work, while the report gets generated asynchronously, starting form step A-2. In this step the ValidationReport is built, but doesn't contain any results yet. This is necessary to have an unique identifier to refer to by the ValidationResults generated later. Also in A-2 all the required textdocuments are loaded and put into the Memcache (see subsection 3.1.3) for later use. During this process meta information about the textdocuments, like the amount of textdocument or number of units can be extracted if needed for the ValidationReport. Then A-2 starts multiple tasks wich perform steps A-3 and A-4.

In step A-3 multiple tasks extract the information about the coded units from the textdocuments (explained in subsection 5.2.3) and prepare the input data for the selected inter-coder agreement metric in step A-4. Step A-4 runs the actual algorithm of the metric with the input data and produces a ValidationResult, which gets stored to the DataStore. Steps A-3 and A-4 try to fetch the textdocuments they need from the Memcache, but access the DataStore if a textdocument already expired from the Memcache.

The steps A-3 and A-4 from Figure 5.1 can be implemented for any feasible inter-coder agreement metric. Furthermore this process reduces the vendor lock-in to the GAE cloud platform as the underlying program logic for the metric from step A-3 and A-4 can be run in any other environment, because of its independency from environmental services such as the DataStore.

In the optional step A-5 an average can be calculated over all the ValidationResults that are part of a ValidationReport. The process waits for all tasks to finish and write their results to the DataStore, before calculating the average. Step A-5 is independent from the used inter-coder agreement metric and simply calculates the average from the generic data structure ValidationResult into ValidationReport wich are explained in subsection 5.2.2.

[1]HTTP/1.1 Status code definitions https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Case B of Figure 5.1 shows accessing a report by the user. In this case the users request can be handled directly by the ValidationEndpoint and the ValidationReport including the ValidationResults is sent to the user directly and gets presented by the frontend.

5.2.2 Report Data Structures

Figure 5.2: Simplified UML Class Diagram of ValidationReport and ValidationResult

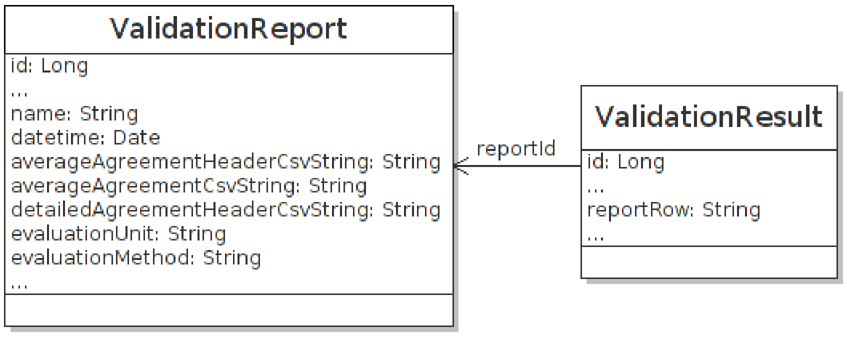
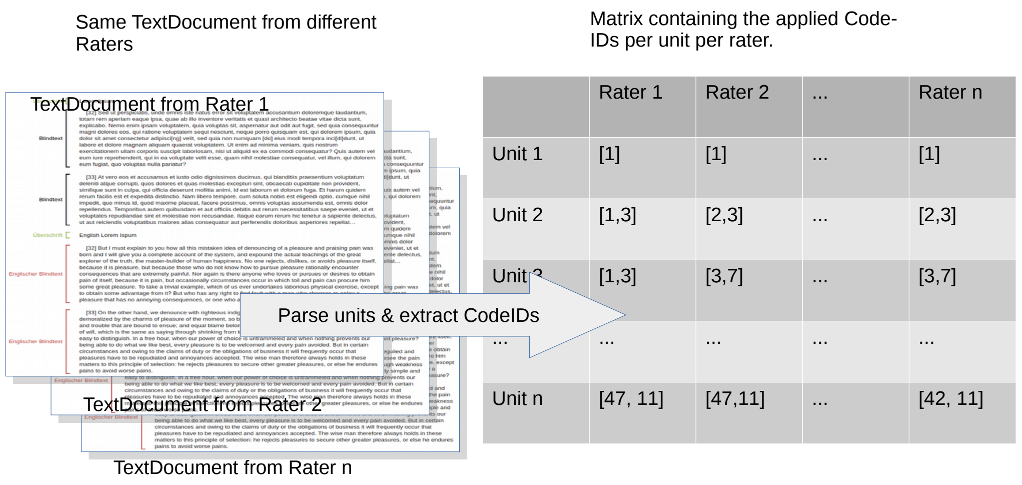


Figure 5.2 shows the ValidationReport and ValidationResult class. In Case A from Figure 5.1 one ValidationReport gets saved to the DataStore (Step A-3) and several ValidationResults can be created by the started tasks (Step A-4) in the TaskQueue. The implemented structures are kept generic in order to be able to save different kinds of results efficiently in the DataStore. The attributes *averageAgreementHeaderCsvString* and *averageAgreementCsvString* of the ValidationReport class (see Figure 5.2) are set and persisted, if an average of all ValidationResult instances for one report is calculated (Step A-5 of Figure 5.1). As the variable names already suggest, the results are saved as Comma Separated Value (CSV) Strings. This allows to persist reports with an arbitrary number of individual results as Strings, which does not restrict the data type for the result. There is also no restriction for the amount of ValidationResult instances that belong to a ValidationReport. Also the *reportRow* attribute in the ValidationResult class gets saved as CSV String and the *detaildAgreementHeaderCsvString* attribute from ValidationReport gives the meta-information on how to interpret the ValidationResult instances. On the QDAcity frontend a tabular view is generated out of the persisted ValidationReport and its ValidationResult instances.

5.2.3 Extraction of Input Data for Quality Metrics

The input data for both implemented inter-coder agreement metrics in this thesis is based on coded units of text by different raters. In QDAcity the information about the coded units is stored in the textdocuments directly as explained in section 4.3.

Figure 5.3: Textdocuments - Units and their codings by different raters



Same TextDocument from different Raters
TextDocument from Rater 1
TextDocument from Rater 2
TextDocument from Rater n
Parse units & extract CodeIDs
Matrix containing the applied Code-IDs per unit per rater.

	Rater 1	Rater 2	...	Rater n
Unit 1	[1]	[1]	...	[1]
Unit 2	[1,3]	[2,3]	...	[2,3]
Unit 3	[1,3]	[3,7]	...	[3,7]
...	...	...	...	...
Unit n	[47, 11]	[47,11]	...	[42, 11]

Figure 5.3 illustrates the general strategy for aggregating the applied codes per unit by rater. After parsing the textdocument and extracting the applied code-IDs from the units (see section 4.3) they get transformed to a three dimensional datastructure, as shown in Figure 5.3 in the table on the right side. For every unit and rater the list of applied codes is saved.

The process from Figure 5.3 is implemented by the *TextDocumentAnalyzer* class. It expects a List of textdocuments, containing the same textdocument, but with the ratings from different raters and the evaluation unit, in order to split the textdocument in the proper units. The output is a three dimensional datastructure, which is the representation for the table on the right in Figure 5.3.

This generic data structure is then used to generate the input data for the algorithms as explained in the implementation details subsections of the next two sections.

Table 5.2: Reliability Data as input for Krippendorff's Alpha

Units u:		1	2	...	N	
Observers:	1	c_{11}	c_{12}	...	c_{1N}	
	2	c_{21}	c_{22}	...	c_{2N}	
	...	...	...	...	...	
	m	c_{m1}	c_{m2}	...	c_{mN}	
# observers valuing	u:	m_1	m_2	...	m_N	$\sum_u m_u$

(Adopted from (Krippendorff, 2011, p. 6))

Table 5.3: Coincidence Matrix for Krippendorff's Alpha

Values	1	...	k	...	
1	o_{11}	...	o_{1k}	...	n_1
...	...	...	...	...	...
c	o_{c1}	...	o_{ck}	...	$n_c = \sum_k o_{ck}$
...	...	...	...	...	...
	n_1	...	n_k	...	$n = \sum_c \sum_k n_{ck}$

(Adopted from (Krippendorff, 2011, p. 6))

5.3 Krippendorff's Alpha Coefficient

The Krippendorff's Alpha Coefficient (Krippendorff, 2011) is a statistical measure for inter-coder agreement. Agreement is measured among raters which categorize units.

The input for the Krippendorff's Alpha coefficient is the so called reliability data, sketched in Table 5.2. c_{mN} is the value c coder m has set to unit N. Note that a coder can only set one value to a unit, as the ratings must be categorical (see section 5.1).

Out of the reliability data the so called coincidence matrix has to be calculated. The coincidence matrix is outlined in Table 5.3. With $c, k \in$ set of categories, the coincidence matrix is always square. The calculation for a matrix entry o_{ck} is the following:

$$o_{ck} = \sum_u \frac{pairs(u)}{m_u - 1} \tag{5.1}$$

Where $pairs(u)$ is the number of c-k pairs in unit u, equals to the number of raters having set category c unit u.

As of (Krippendorff, 2011, p. 1f) the general form of α is the following:

$$\alpha = 1 - \frac{D_o}{D_e} \tag{5.2}$$

Table 5.4: Interpretaion of Krippendorff's Alpha results

Krippendorff's Alpha value	Reliability
$\alpha = 1$	Perfect reliability
$0 < \alpha < 1$	Percentage reliability
$\alpha = 0$	No reliability (assinged values to units are statistically unrelated)
$\alpha < 0$	systematic disagreement D_o exceeding disagreement expected by chance D_e.

with D_O as the observed disagreement of the given ratings and D_e as the disagreement expected by chance. They are defined as:

$$D_o = \frac{1}{n} \sum_c \sum_k o_{ck} \;\; {}_{metric}\delta^2_{ck} \tag{5.3}$$

$$D_e = \frac{1}{n(n-1)} \sum_c \sum_k n_c \times n_k \;\; {}_{metric}\delta^2_{ck} \tag{5.4}$$

Where ${}_{metric}\delta$ is the used difference function depending on the data. As stated in Table 5.1 Krippendorff's Alpha supports binary, nominal, ordinal, interal, ratio, polar and circular data. The data which is coming from coded units of texts in QDAcity is nominal and therefore, in this thesis, only the nominal difference function of the following form is needed (Krippendorff, 2011, p. 5):

$${}_{nominal}\delta^2_{ck} = \begin{cases} 0 \text{ iff } c = k \\ 1 \text{ iff } c \neq k \end{cases} \tag{5.5}$$

5.3.1 Interpretation of Krippendorff's Alpha results

With the basic definition of Krippendorff's Alpha from Equation 5.2, the results can be interpreted as listed in Table 5.4. This interpretation is universally adoptable as it is based on the mathematical definition. However, for the percentage reliability of α no universal interpretation guideline can be given. In QDAcity users can freely define their own interpretation for percentage reliability.

5.3.2 Implementation Details

As mentioned above, Krippendorff's Alpha Coefficient is for measuring the agreement for a categorical rating. However, in QDAcity, more than one code can be applied to a textual unit by one rater at the same time. Therefore, the coding of

Figure 5.4: Simplified UML Class Diagram of AlgorithmInputGenerator and classes

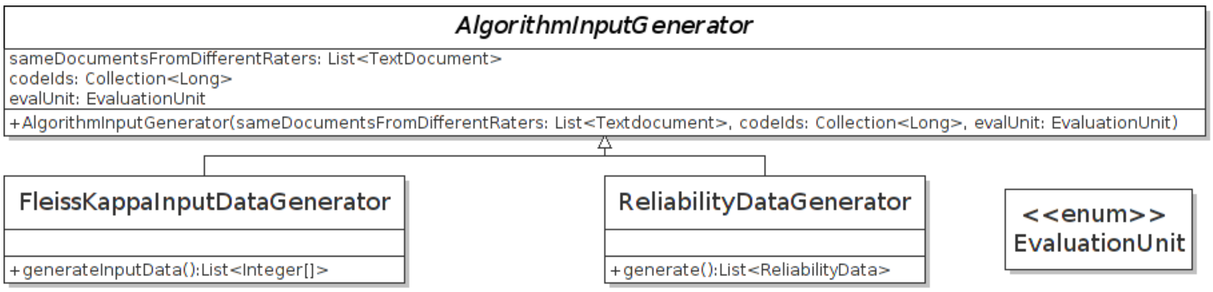
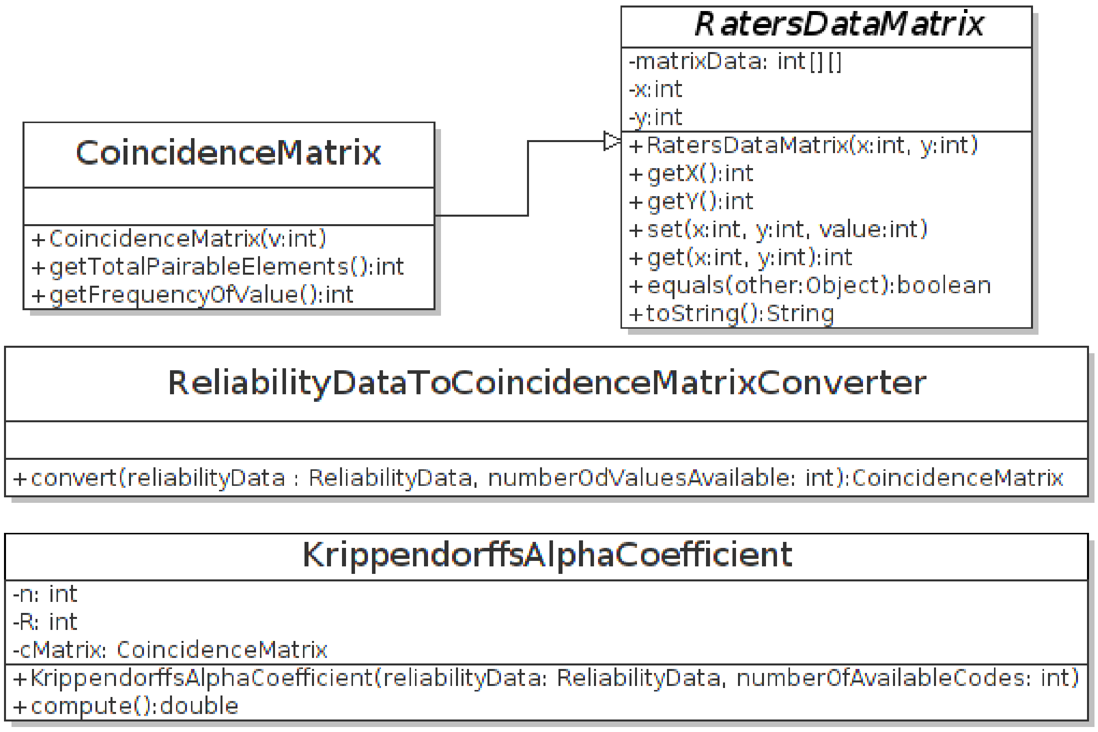

AlgorithmInputGenerator
sameDocumentsFromDifferentRaters: List<TextDocument>
codeIds: Collection<Long>
evalUnit: EvaluationUnit
+AlgorithmInputGenerator(sameDocumentsFromDifferentRaters: List<Textdocument>, codeIds: Collection<Long>, evalUnit: EvaluationUnit)
FleissKappaInputDataGenerator
+generateInputData():List<Integer[]>
ReliabilityDataGenerator
+generate():List<ReliabilityData>
<<enum>>
EvaluationUnit

Figure 5.5: Simplified UML Class Diagram of KrippendorffsAlphaCoefficient and related classes


RatersDataMatrix
-matrixData: int[][]
-x:int
-y:int
+RatersDataMatrix(x:int, y:int)
+getX():int
+getY():int
+set(x:int, y:int, value:int)
+get(x:int, y:int):int
+equals(other:Object):boolean
+toString():String
CoincidenceMatrix
+CoincidenceMatrix(v:int)
+getTotalPairableElements():int
+getFrequencyOfValue():int
ReliabilityDataToCoincidenceMatrixConverter
+convert(reliabilityData : ReliabilityData, numberOdValuesAvailable: int):CoincidenceMatrix
KrippendorffsAlphaCoefficient
-n: int
-R: int
-cMatrix: CoincidenceMatrix
+KrippendorffsAlphaCoefficient(reliabilityData: ReliabilityData, numberOfAvailableCodes: int)
+compute():double

a textual unit in QDAcity is not categorical. In order to apply the Krippendorff's Alpha Coefficient to a coded textdocument (section 4.3) every code of the code system needs to be examined separately. Thus two new categories for every code, which are "Code is set" and "Code is not set" were introduced. An analysis by code now becomes categorical and therefore, the Krippendorff's Alpha Coefficient can be applied. Because of that Krippendorff's Alpha in QDAcity has to be calculated for every single code in the code system for the selected textdocuments. Therefore, a task for every textdocument to be analyzed is run in the TaskQueue. Every task produces one ValidationResult containing the name of the analyzed textdocument and the results of Krippendorff's alpha for every code of the code system. The reliability data for the Krippendorff's Alpha calculations therefore is generated for every code using only two categories.

Figure 5.5 shows the class diagram of the *KrippendorffsAlphaCoefficient* class, which implements the Krippendorff's Alpha coefficient using the nominal difference function. The input is a *ReliabilityData* instance and the number of available codes. The reliability data matches the form as shown in Table 5.2 and gets transferred to the form outlined in Table 5.3 using the *ReliabilityDataToCoincidenceMatrixConverter* class listed in Figure 5.5.

The *ReliabilityData* instance is generated by extracting the information from the textdocuments using the *ReliabilityDataGenerator* class as shown in Figure 5.4. It implements the process explained in subsection 5.2.3.

A ValidationReport with Krippendorff's Alpha in QDAcity also contains an average of all ValidationResults as step A-5 of Figure 5.1 is executed.

5.3.3 Discussion of Krippendorff's Alpha implementation

The approach of using artificial categories as described above also resolves another issue of inter-coder agreement metrics, which is incomplete or missing data. In QDAcity an understanding of incomplete or missing data could be for example, a coder who did not work on the textdocuments yet, at the time when the inter-coder agreement metric is applied. In this case all codes just fall in the category "Code is not set" for this coder.

Another benefit for the user of this implementation is, that a ValidationReport which was calculated with the Krippendorff's Alpha metric contains detailed information about the agreement for every single code. The codes which need further clarification or need to be redefined can be identified using that information.

5.4 Fleiss' Kappa

The first idea for providing users a Kappa statistics in QDActiy was to implement Cohen's Kappa (Cohen, 1960). However, the process of mapping the coded textdocuments (section 4.3) to input data for Cohen's Kappa would alter the data in such way the result is very hard to understand for users of QDAcity. This problem is also discussed in (Powers, 2012). Furthermore Cohen's Kappa only allows to measure the agreement between two raters, which does not match the common use cases of QDAcity (see section 1.2). Therefore, I chose to implement Fleiss' Kappa (Fleiss, 1971) to QDAcity, which is a generalization of Scott's Pi (Scott, 1955), a related statistic to Cohen's Kappa. Fleiss' Kappa allows to calculate the agreement between many raters which assign categorical ratings.

As defined in (Fleiss, 1971, p. 378f) the general form of Fleiss' Kappa is:

$$\kappa = \frac{\overline{P} - \overline{P}_e}{1 - \overline{P}_e} \tag{5.6}$$

Where the divident $\overline{P} - \overline{P}_e$ is the degree of agreement achieved above chance and the divisor $1 - \overline{P}_e$ is the degree of agreement potentially possible above chance.

$\overline{P}$ and $\overline{P_e}$ are defined as:

$$\overline{P} = \frac{1}{N}\sum_{i=1}^{N} P_i \qquad\qquad \overline{P_e} = \sum_{j=1}^{k} p_j^2 \tag{5.7}$$

With N as the total number of subjects and k as the number of categories.

Define n as the number of ratings per subject, $i = 1, ...N$ as subject index and $j = 1, ...k$ as category index. Then n_{ij} becomes the number of raters for subject i in category j. With that P_i and p_j from Equation 5.7 can be defined:

$$p_j = \frac{1}{Nn}\sum_{i=1}^{N} n_{ij} \qquad\qquad \text{note: } \sum_{j=1}^{k} p_j = 1 \tag{5.8}$$

$$P_i = \frac{1}{n(n-1)}\sum_{j=1}^{k} n_{ij}(n_{ij} - 1) \tag{5.9}$$

The general structure of the input data for Fleiss' Kappa is shown in Table 5.5 using the definitions from the equations above.

Table 5.5: General form of input data for Fleiss' Kappa

	1	2	...	k	
1	n_{11}	n_{12}	...	n_{1k}	P_1
2	n_{21}	n_{22}	...	n_{2k}	P_2
...	...	...	...	...	...
N	n_{N1}	n_{N2}	...	n_{Nk}	P_N
$\sum$	$\sum_{i=1}^{N} n_{i1}$	$\sum_{i=1}^{N} n_{i2}$	...	$\sum_{i=1}^{N} n_{ik}$	
	p_1	p_2	...	p_k	

Table 5.6: Interpretaion of Fleiss' Kappa results

Fleiss' Kappa value	Strength of Agreement
<0.00	Poor
0.00-0.20	Slight
0.21-0.40	Fair
0.41-0.60	Moderate
0.61-0.80	Substantial
0.81-1.00	Almost Perfect

(Adopted from (Landis & Koch, 1977, p. 165))

5.4.1 Interpretation of Fleiss' Kappa

As suggested in (Landis & Koch, 1977, p. 165) the result of Fleiss' Kappa could be interpreted as listed in Table 5.6.

However, the suggested interpretation can not be universally applied to every context. Therefore, this interpretation is not adopted to QDAcity in order to allow users to freely define their own interpretation for the raw Fleiss' Kappa result.

From a mathematical point of view, only the values $\kappa = 1$ as complete agreement and $\kappa \leq 0$ as no agreement beyond the agreement by chance can be defined in a general sense.

For interpreting the result of Fleiss' Kappa also other qualitites of the input data have to be taken into account as discussed in (Sim & Wright, 2005) in detail. The number of categories has influence on the likelyhood the result of Fleiss' Kappa is higher or lower. More categories can potentially lead to more disagreement, therefore fewer categories result in a higher κ (details see (Sim & Wright, 2005, p. 264)).

5.4.2 Implementation Details

As for Krippendorff's Alpha Coefficient also for Fleiss' Kappa agreement only for categorical ratings can be calculated. The same strategy as for Krippendorff's Alpha Coefficient applies here to break down the code-system based ratings to categories, by looking at the codes one by one and introducing "Code is set" and "Code is not set" as categories.

For computing the Fleiss κ only an array of *Integer* and the number of raters as input is needed. Because of breaking down the categories of codes as explained above, the only information needed to calculate Fleiss' Kappa is the number of raters who applied the code in a unit and the number of raters.
The array of Integer is generated by the FleissKappaInputDataGenerator class as shown in Figure 5.4, which uses the process from Figure 5.3 and transforms it to the form outlined in Table 5.5.

Due to the artificially introduced categories for codes they have to be analyzed separately, similar to the Krippendorff's Alpha implementation as described in subsection 5.3.2. Therefore, for every textdocument to be analyzed a task is put into the task queue, performing step A-3 and step A-4 of Figure 5.1. The *FleissKappaInputDataGenerator* class from Figure 5.4 extracts a Integer array for every code of the code system from the same textdocuments from different rates for this task as described above. The *FleissKappa* class calculates the κ for every code and when all results have been calculated they are written into a *ValidationResult* (see Figure 5.2).

In addition, the implementation for Fleiss' Kappa performs step A-5 of Figure 5.1 and calculates the average for every code from all ValidationResults of the report.

5.4.3 Discussion of Fleiss' Kappa implementation

The benefit for the user of this implementation is the possibility to check all the codes one by one, which allows to identify codes, which need more clarification or need to be redefined.

However, as explained in subsection 5.4.1, fewer categories in the Fleiss' Kappa statistic lead to a higher agreement. Therefore, the results calculated in QDAcity should be interpreted taking into account, that they are only based on two categories per code.

6 Saturation

In this chapter, the approach and the implementation to question (b) of chapter 2 is presented and discussed. First, an overview of related work is given, and then a definition of saturation in the context of QDAcity is given, and at last the relevant details of its implementation are highlighted.

6.1 Related Work

As shown in Table 6.1, three different forms of saturation exist. These are based on two different approaches, which are the grounded theory ((Corbin & Strauss, 1990), (Glaser, Strauss, & Strutzel, 1968)) and the idea that the results evolve when new data is added ((Fusch & Ness, 2015), (Kerr, Nixon, & Wild, 2010)). Nevertheless these three forms of saturation are closely related and mostly differ in the way saturation is measured as listed in Table 6.1.

An example for data and thematic saturation is presented and discussed in (Guest et al., 2006). Using an example project the authors outline when data saturation is reached. The authors address both data and thematic saturation, but only use the term data saturation for their saturation result. An example for theoretical saturation explained and discussed in (Bowen, 2008) in detail.

However, literature review revealed that most of the authors define saturation in vague natural language and do not provide a transparent formula or specific criteria which show the details of their saturation calculation. This issue is also discussed in (O'Reilly & Parker, 2013) and (Bowen, 2008).

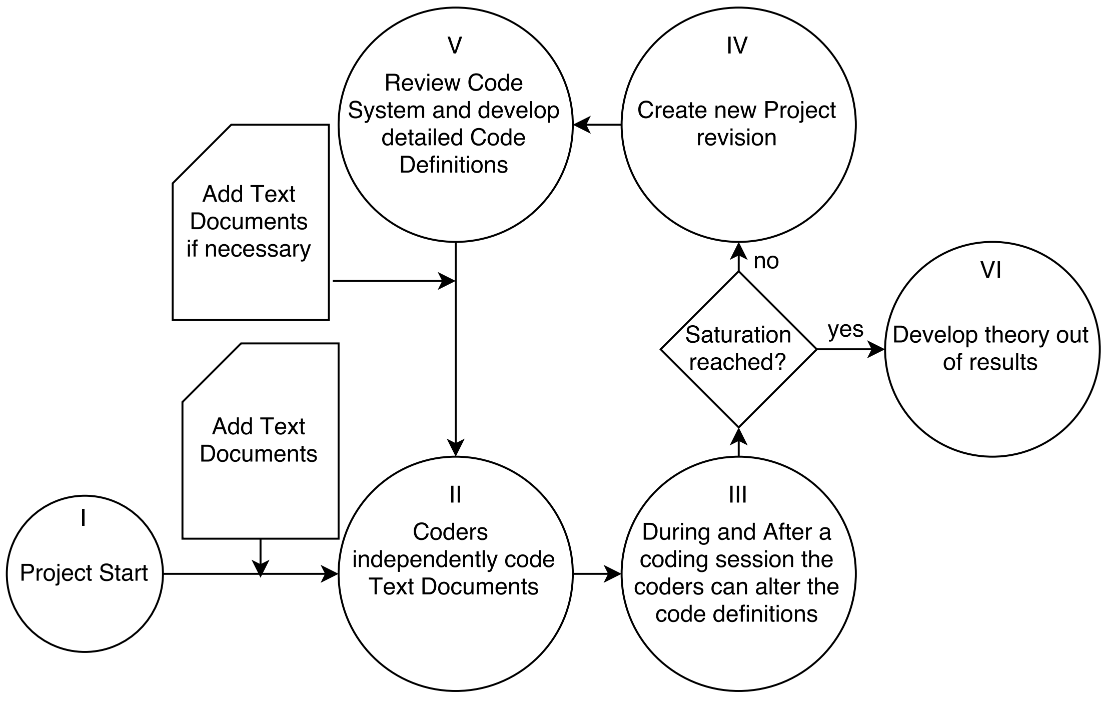
6.2 Definition of Saturation in QDAcity

The idea for saturation in QDAcity projects is that the activity on the code system (Figure 4.2) decreases over time, until it is stable enough to form a theory on it. This notion of saturation is based on the recommended working process

Table 6.1: Forms of saturation

Name & Citations	Based on the idea ...	Measure
Theoretical Saturation (Guest, Bunce, & Johnson, 2006) (MacQueen, McLellan, Kay, & Milstein, 1998)	... of a grounded theory.	Decrease of **activity on code system**. "Stability of code system"
Data Saturation (Francis et al., 2010)	... that results are based on the data.	Decrease of **new insights** from new data.
Thematic Saturation (Guest, Bunce, & Johnson, 2006)	... that results are based on the data.	Decrease of **new topic exploration** with new data.

Figure 6.1: Outline of general working process in a QDAcity project



V
Review Code System and develop detailed Code Definitions
IV
Create new Project revision
Add Text Documents if necessary
no
VI
Saturation reached?
yes
Develop theory out of results
Add Text Documents
I
Project Start
II
Coders independently code Text Documents
III
During and After a coding session the coders can alter the code definitions

for QDAcity projects outlined in Figure 6.1. After the project start in step I usually the project owner can set up a first inital code system which is just an outline, to get a first idea or could even be empty for free coding. Furthermore, the first textdocuments have to be added to be able to start coding. Step II and III usually take place at the same time as only through the process of coding the coding system can be improved.

After step II of Figure 6.1 the code system could either be stable enough to finish the project and develop a theory (step VI), or if the code system is not finished, the project continues. In this case, it is recommended to create a new ProjectRevision (see Figure 4.1) and then add new textdocuments if necessary (or existent) to repeat the process from step II.

Derived from the working process of Figure 6.1 the implemented saturation feature in this thesis can be categorized as theoretical saturation, because it is based on a grounded theory and measures the activity on the code system (see Table 6.1). The key question to answer by the saturation feature is: When is the code system stable enough to consider the project as finished?

6.3 Technical approach

In this section, the relevant technical details of the implemented saturation feature in QDAcity are illustrated.

6.3.1 Logging of Changes

In order to be able to apply a theoretical saturation metric, there needs to be a mechanism to access all changes on the code system from a time period. Therefore, the changes need to be logged and persisted to the DataStore (subsection 3.1.2). Table 6.2 shows the Change Objects and the Change Types which are persisted. The combination of Change Object and Change Type and changed attribute is the category c of a persisted Change. Additionally change groups are defined, which are defined by combination of Change Object and Change Type. The change group of a change category c is noted down as $\hat{c}$. All categories form the set of change categories $\mathbb{K}$.
The potential change combinations derive from the changeable attributes of codes, codebook entries and code relations as described in Figure 4.2 and the way applied codes are stored in textdocuments as explained in section 4.3.

Furthermore $\mathbb{C}$ is defined as the set of all persisted changes. As in one change operation multiple attributes of a Change Object can be changed (see Table 6.2, CODE and CODEBOOK_ENTRY) a change can be from multiple categories, but only from one change group.

6.3.2 Calculation of Saturation

As discussed in section 6.1 no formula for the calculation of saturation can be adopted from previous work. Therefore, in this thesis a completely new formula for calculating the saturation S has been developed following the notion of theoretical saturation from previous work (section 6.1). In the following instead of the term *theoretical saturation* the term *saturation* is used.

Saturation is always calculated over a specific time period. Define $t1 > 0$ as the starting point and $t2 \geq t1$ as the end point of the saturation's period. Furthermore saturation is calculated for a specific Category C of Changes.
In general saturation of a category C in the time period $t1$ to $t2$ is defined as:

$$S^{C}_{t1,t2} = 1 - A^{C}_{t1,t2} \tag{6.1}$$

Where $A^{C}_{t1,t2}$ is the activity in the category C in the time period $t1$ to $t2$ and is defined as

$$A^{C}_{t1,t2} = |\mathbb{C}^{C}_{t1,t2}| / |\mathbb{C}^{\hat{C}}_{0,t1}| \tag{6.2}$$

where $\mathbb{C}^{C}_{t1,t2}$ is the set of changes in the time period $t1$ to $t2$ in the category C. Note that in the divisor of Equation 6.3.2 all changes from the change group $\hat{C}$ (see subsection 6.3.1) of the category C are counted from 0 until t_1.

6.3.2.1 Weighted Average of Saturation

As shown in the section before, $S^{C}_{t1,t2}$ needs to be calculated for every possible change type which - in QDAcity - results in 16 single saturation values. However, question (b) from chapter 2 and section 2.3 indicate that a single-valued and configurable measure for the project proceeding is a great benefit of the user. Therefore, the weighted average of the saturation $S^{*}_{t1,t2}$ was introduced in QDAcity and is defined as

$$S^{*}_{t1,t2} = \sum_{c \in \mathbb{K}} S^{c}_{t1,t2} * \left(W_c / \sum_{c2 \in \mathbb{K}} W_{c2} \right) \tag{6.3}$$

where $\mathbb{K}$ is the set of categories[1] of changes as described in subsection 6.3.1. $0 \leq W_c \leq 1$ is the weight of the category of changes. The weighting can be freely

[1]In this thesis, changes with the change object DOCUMENT as listed in Table 6.2 are logged, but not part of the saturation calculation as it does not affect the code system and therefore is not part of theoretical saturation.

Table 6.2: Tracked Change Objects and their possible Change Types in QDAcity

Change Object	**Change Type**	**Used when ...**
CODE	CREATED	a new Code is created in the CodeSystem.
	MODIFIED	one or more of the following Code's attributes are changed: author, color, memo, name, subCodeIDs
	RELOCATE	a Code gets moved and therefore has a new parent.
	DELETED	a Code gets deleted from the CodeSystem.
CODEBOOK_ENTRY	MODIFIED	a Code's CodeBookEntry is modified. The following atirubtes can be changed: definition, example, shortDefinition, whenNotToUse, whenToUse
CODE_RELATIONSHIP	CREATED	a new relation between codes is introduced.
	DELETED	a relation between codes gets removed.
DOCUMENT	CREATED	a textdocument gets created.
	APPLY	a Code gets applied in the textdocument.

Table 6.3: Default weighting of change categories in QDAcity

Name	**Weight**
Code Changes	
insertCodeChangeWeight	1.0
updateCodeAuthorChangeWeight	0.0
updateCodeColorChangeWeight	0.0
updateCodeMemoChangeWeight	0.5
updateCodeNameChangeWeight	1.0
relocateCodeChangeWeight	1.0
deleteCodeChangeWeight	1.0
Code Relations	
insertCodeRelationShipChangeWeight	0.75
deleteCodeRelationShipChangeWeight	0.75
CodeBookEntry Changes	
updateCodeBookEntryDefinitionChangeWeight	1.0
updateCodeBookEntryExampleChangeWeight	0.75
updateCodeBookEntryShortDefinitionChangeWeight	1.0
updateCodeBookEntryWhenNotToUseChangeWeight	0.75
updateCodeBookEntryWhenToUseChangeWeight	0.75

defined by the user. In QDAcity a preset of weightings is implemented as shown in Table 6.3. Further parameters, which also influence the weighted average of saturation are described in the next section.

6.3.3 Saturation Parameterization

In order to meet common markers of quality in qualitative research (see (Spencer, Ritchie, Lewis, & Dillon, 2003)), the calculation of saturation in QDAcity is user configurable and therefore more transparent for the user. Beside the weighting of the single categories from the section above, the user can configure two additional aspects of the saturation calculation.

An idea of the saturation feature in QDAcity is to compute saturation regularly and see its historical developement over time. Therefore, the period of the saturation calculation can be parameterized as follows: In QDAcity t_2 as used in the equations above is always set to the time of execution of a saturation calculation. In order to get a gapless historical developement of the saturation, t_1 always gets set to the end time t'_2 of an historical saturation calculation result. This does not necessarily need to be the latest saturation calculation, but could also be the second or third last saturation result. The user can set this freely between latest

Table 6.4: Default maximum saturations of change categories in QDAcity

Category	**Default Maximum Saturation**
Code Changes	
Insert Code	1.0
Update Code Author	0.1
Udate Code Color	0.1
Update Code Memo	0.9
Update Code Name	1.0
Relocate Code	1.0
Delete Code	1.0
Code Relations	
Insert Code Relationship	0.95
Delete Code Relationship	0.95
CodeBookEntry Changes	
Update CodeBookEntry Definition	1.0
Update CodeBookEntry Example	0.9
Update CodeBookEntry Short Definition	1.0
Update CodeBookEntry When Not To Use	0.9
Update CodeBookEntry When To Use	0.9

or 20^{th} last saturation result[2]. If no previous or n^{th} last saturation calculation exist, t_1 is set to beginning of Unix time[3], which is always before any change could have persisted in a QDAcity project.

The second aspect to be parameterized affects the interpretation of a saturation result. 100% of saturation means complete saturation. However, in reality this value is hard to achieve and therefore users might have an understanding, that a percentage value below 100% is already complete saturation in their context. Thus users are able to set the maximum in each change category which is their sufficient saturation. The saturation result then gets normalized using the configured maximum of each category using the following formula:

$$|S^C_{t1,t2}| = min\left(1, \frac{S^C_{t1,t2}}{m_C}\right) \tag{6.4}$$

Where m_C is the configured maximum saturation in the category C with $0 < m_C \leq 1$. $min(x, y)$ is the minimum function, resulting in x if $x \leq y$ and y otherwise.

[2]The limit was set to maximum 20 last saturation result in order to avoid excessive computing time consumption for results with small expressiveness. The default setting is 3.

[3]using java.util.date with *new Date(0)* resulting in 1^{st} January 1970 00:00:00

With the equation above also the weighted average of saturation is normalized as follows:

$$|S^{*}_{t1,t2}| = \sum_{c\in\mathbb{K}} |S^{c}_{t1,t2}| * \left(W_c / \sum_{c2\in\mathbb{K}} W_{c2}\right) = \sum_{c\in\mathbb{K}} min\left(1, \frac{S^{c}_{t1,t2}}{m_c}\right) * \left(W_c / \sum_{c2\in\mathbb{K}} W_{c2}\right) \tag{6.5}$$

QDAcity comes with a preset maximum saturation for all categories as shown in Table 6.4.

6.3.4 Implementation Details

Figure 6.2: Simplified UML class diagram of SaturationCalculator and related classes

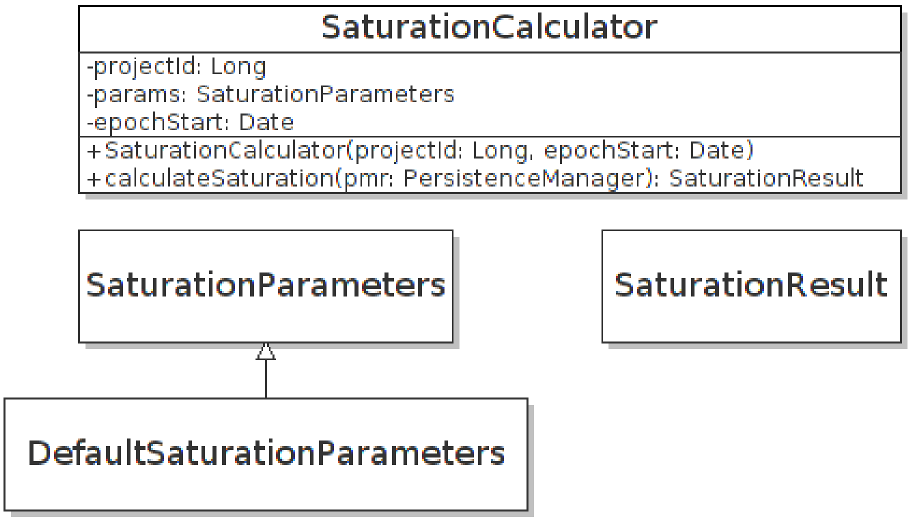
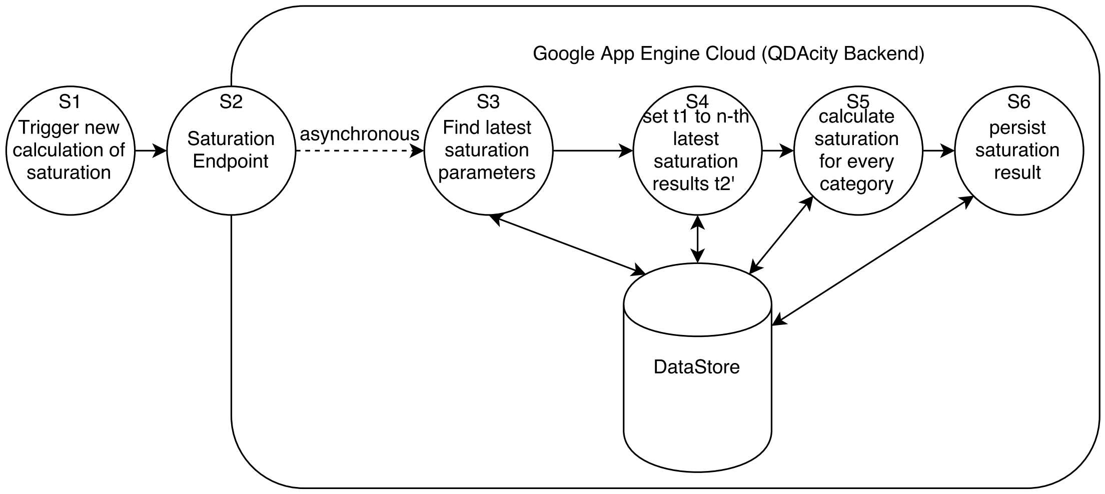

SaturationCalculator
-projectId: Long
-params: SaturationParameters
-epochStart: Date
+SaturationCalculator(projectId: Long, epochStart: Date)
+calculateSaturation(pmr: PersistenceManager): SaturationResult
SaturationParameters
SaturationResult
DefaultSaturationParameters

Figure 6.2 shows the *SaturationCalculator* class, which calculates a *SaturationResult*, containing all 16 single saturation values from the different categories. For the calculation the *SaturationCalculator* needs the *projectId* in order to access all the persisted changes from the project and the *epochStart*, which matches t_1 from the equations in subsection 6.3.2.

The *calculateSaturation* method expects an instance of *PersistenceManager*, which is used for accessing the DataStore. It is necessary, that the caller of *SaturationCalculator.calculateSaturation(...)* is using the same instance of *PersistenceManager*, because the *SaturationCalculator* needs to persist the *SaturationResult* before calculation in order to be able to add the used *SaturationParameters* as child element and the caller is responsible for persisting the complete result again. This is only possible with the same *PersistenceManager* instance as another instance would generate another id for the SaturationResult in the DataStore.

The *SaturationParameters* class as shown in Figure 6.2 contains the configuration for the saturation calculation as explained in subsubsection 6.3.2.1 and subsection 6.3.3. The *DefaultSaturationParameters* contains the preset listed in Table 6.3 and Table 6.4 as well as the default configuration for setting t_1, which is t'_2 of the 3^{rd} last saturation calculation. By checking the existing SaturationResults in the DataStore for the project, the value for t_1 is determined by the caller of *SaturationCalculator.calculateSaturation(...)* and set for *epochStart*.

Figure 6.3: Process of calculating a new SaturationResult in QDAcity


Google App Engine Cloud (QDAcity Backend)
S1 Trigger new calculation of saturation
S2 Saturation Endpoint
asynchronous
S3 Find latest saturation parameters
S4 set t1 to n-th latest saturation results t2'
S5 calculate saturation for every category
S6 persist saturation result
DataStore

The overall process of calculating a new *SaturationResult* in QDAcity is shown in Figure 6.3. In step S1 a new saturation calculation is triggered on the frontend. There are different types of triggers as explained in subsubsection 6.3.4.1. The frontend sends a request to the SaturationEndpoint (step S2) which asynchronously pushes a new task to a TaskQueue for calculating the saturation and then immediately replies with a HTTP 204 "No Content". In the started task steps S3 to S6 are performed. First, the latest saturation parameters have to be loaded from the DataStore (step S3) in order to get the configuration from which n^{th} latest SaturationResult t_1 can be set (step S4). After that the actual saturation is calculated for every change category in step S5, accessing the persisted changes form the DataStore. At last the SaturationResult has to be persisted in the DataStore in step S6.

6.3.4.1 Triggering Saturation Calculations

Derived from the recommended working process in QDAcity as shown in Figure 6.1 and detailly described in section 6.2, two kinds of triggers for calculating a new saturation have been implemented for QDAcity in this thesis.

The first trigger is linked to the creation of a new ProjectRevision (step VI in Figure 6.1). When a user creates a new ProjectRevision it automatically triggers the process from Figure 6.3 and a new SaturationResult is generated.

The second trigger is linked to steps IV and V of Figure 6.1. While users code the textdocuments they are also redefining the code system. Consequently DOCUMENT-APPLY-Changes as listed in Table 6.2 come along more changes from the other categories. Therefore, these changes can be used as a trigger for a new saturation calculation. In QDAcity after every 10 changes of the group DOCUMENT-APPLY a new saturation calculation as outlined in Figure 6.3 is triggered.

6.3.4.2 Latency Optimization

Creating a deferred task for TaskQueues (subsection 3.1.4) for work which could have a negative impact on user experience in terms of latency is a common strategy on GAE. Therefore, it is particularily interesting from which point on, the creation of a new task is faster than doing the work directly.

As explained in subsection 6.3.1, logging of changes is essential for the calculation of saturation. However, as every change on the codesystem, which is triggered by a user action needs to be logged the time needed for logging[4] should not have a negative impact on the user experience regarding reaction time.

In this thesis, two alternative methods for change logging have been implemented and tested. The first method is logging the changes directly within the users request. The user only gets a response after the change has been logged. The second method creates a deferred task for logging the change and the change gets persisted into the DataStore asynchronously. The user gets a response after the task for logging the change has been created.

In an automated test, which sent an update code request every 600ms to one QDAcity instance[5] the two implementations were compared. The test was performed on an F1 instance (see (Google Inc., 2017b) and (Google Inc., 2017a)). In a first run only the direct logging was used and in a timely separated second run only the deferred logging was used. Figure 6.4 shows how many percent of update code calls had which latency. Latency in this figure is defined by the time passed from recieving the user's request to sending the response. The blue line shows the percentages for the method with direct logging, the orange line

[4]The time needed for logging includes (1) creation of the change as Java object (2) Putting the change into the DataStore

[5]In a manual test an updateCode call lasted around 500-600ms. By timely separating the requests, I avoided side effects of scheduling, which affect the latency.

Figure 6.4: Comparison of latency of update code calls with deferred and direct change logging on F1 instance

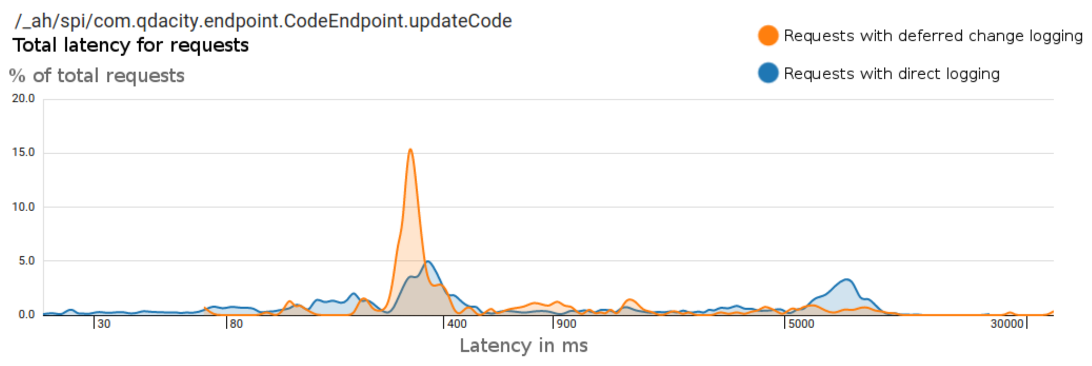


Figure 6.5: Comparison of cumulative latency of update code calls with deferred and direct change logging on F1 instance

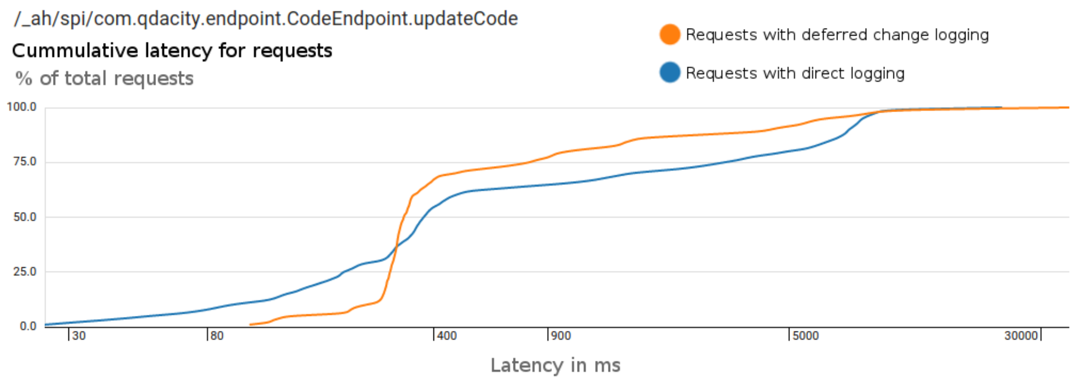


Figure 6.6: Comparison of latency of update code calls with deferred and direct change logging on F4 instance

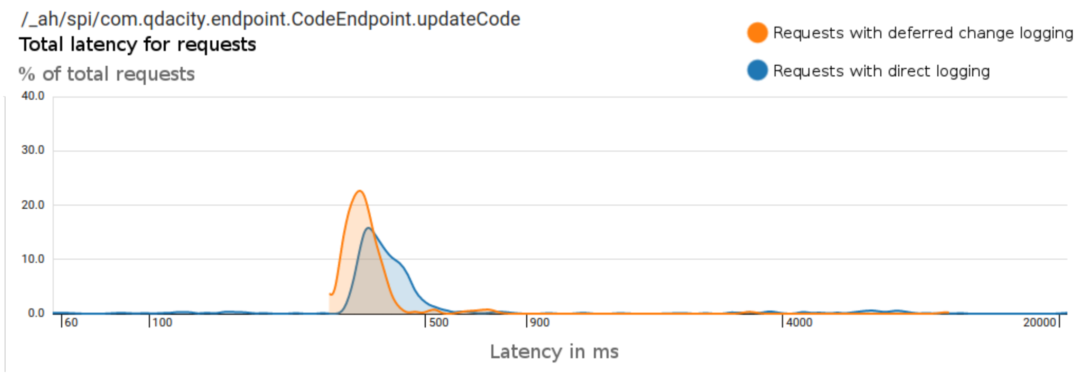
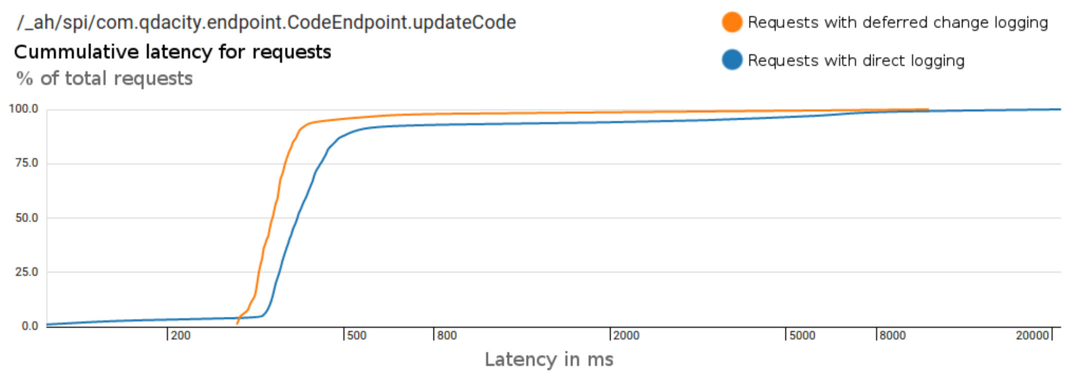


Figure 6.7: Comparison of cumulative latency of update code calls with deferred and direct change logging on F4 instance



for deferred logging. In Figure 6.5 the accumulated percentage for the latency is shown based on the same data and using the same colors.

Derived from the graphs in Figure 6.4 and Figure 6.5 the implementation using deferred change logging is faster as most of the requests have a latency less than 900ms. Furthermore, when using direct logging there is a higher percentage of requests which need more than 5000ms, which has a direct and negative impact on the user experience.

The test was repeated on a F4 GAE instance (see (Google Inc., 2017b) and (Google Inc., 2017a)). Figure 6.6 and Figure 6.7 show the latencies as explained for Figure 6.4 and Figure 6.5 using the same colors. Also on a F4 instance the deferred change logging leads to a shorter latency which is better for the user's experience.

Derived from the performed tests above, I selected the implementation using deferred change logging to achieve shorter latency for changes on the code system.

6.3.5 Discussion of saturation feature

The presented saturation feature uses change categories as explained in subsubsection 6.3.2.1. However, the mathematical formula for calculating the saturation is not dependend on the specific categories, types and attributes shown in Table 6.3 and thereforethe presented saturation feature is not restricted to the use case of QDAcity.

The weighting and the maximum for the single categories of saturation are freely user configurable as described in subsubsection 6.3.2.1 and subsection 6.3.3. Therefore, users can set parameters which fit their use case. However, I do not provide a guideline on how to properly set the parameters and there are no restrictions for users to also set bogus parameter combinations.

7 Evaluation

In this chapter, it is checked whether the requirements defined in section 2.3 are met. Secondly, the tests performed to prove the correctness and usefulness of the implemented features are presented.

7.1 Requirements

The requirements listed in section 2.3 from the problem definition are fulfilled by the presented implementation in this thesis as listed below:

1. The results of inter-coder agreement metrics contain a single value representing the average, the agreement by code and agreement by rater.
2. As described in subsection 5.2.1 the architecture for inter-coder agreement metrics is extensible and provides a generic environment so that any feasible inter-coder agreement metric can be implemented. Storage and representation of the agreement are abstracted from the used algortihms as explained in subsection 5.2.2.
3. The saturation metric shows the single-valued weighted average on the project's dashboard on the frontend of QDAcity and allows to view the detailed values for every component in a tabular view.
4. The saturation feature shows the historical developement of saturation in a line diagram via the project's dashboard on the frontend.
5. The saturation metric allows the user to configure the weights of the single components in the average saturation and the maximum needed by a single component to reach saturation.
6. A preset for the weighting (Table 6.3) and the saturation maxima (Table 6.4) is provided.
7. The architecture for inter-coder agreement metrics and the saturation metric implements the actual algorithms in separate classes, which can be used

in any other environment as they have no dependencies to GAE.

7.2 Correctness of Quality Metrics

As described in chapter 5, two inter-coder agreement metrics have been adopted and implemented. In order to prove that specific steps of these metrics are correct four JUnit tests have been set up.

7.2.1 JUnit Tests for Krippendorff's Alpha

There are three JUnit tests implemented for Krippendorff's Alpha.

The first JUnit test tests the correctness of the *KrippendorffsAlphaCoefficient* class, which is calculating the Krippendorff's α out of a given reliability data. The result is compared to a manual calculation of the α value for the given reliability data using Java assertions[1].

The second JUnit test proves the correctness of the implemented process for converting the reliability data (see Table 5.2) to the coincidence matrix (see Table 5.3).

The third JUnit test checks if the result of extracting the reliability data out of the textdocuments (see subsection 5.2.3). A mockup for two *TextDocument* instances containing codings from different raters has been implemented to use it with this JUnit test. The expected reliability data has been implemented as mockup as well by manually conducting the process from subsection 5.2.3 on the two *TextDocument* instances. Via the *equals*() function on a *ReliabilityData* instance, it is possible to compare it with another *ReliabilityData* instance as the *equals*() method has been overridden in order to compare every single value in the reliability data. Using Java assertions the JUnit test proves that the generated reliability data is equal to the manually created.

All three JUnit tests implemented for Krippendorff's Alpha run without any errors or warnings.

[1]Using Assertions in Java Technology http://www.oracle.com/us/technologies/java/assertions-139853.html

7.2.2 JUnit Test for Fleiss' Kappa

The JUnit test for Fleiss' Kappa tests the *FleissKappa* class, which calculates the κ value out of the given input data outlined in Table 5.5 and the number of raters.

The κ value for the example in the JUnit test has been manually calculated and gets compared with the value calculated by the *FleissKappa* class using Java assertions.

The implemented JUnit test for Fleiss' Kappa passes without any errors or warnings.

7.3 User's benefit

In order to integrate the new inter-coder agreement feature to QDAcity the given implementation accessing the F-Measure through the QDAcity frontend was adopted and expanded to the new inter-coder agreement metrics. The new inter-coder agreement metrics have been tested manually using the QDAcity frontend on mock-up projects covering the QDAcity use cases (section 1.2). The mock-up projects contained large textdocuments and up to four coders. Additionally, large coding systems with up to 25 codes were used during the tests, in order to verify the readability of the result tables of the inter-coder agreement metric. The tests were conducted in projects containing codings with good agreement and complete disagreement from chaotic data to verify the behaviour of the implemented inter-coder agreement metrics in these cases.

The mock-up projects described above were set up using the process outlined in Figure 6.1. Therefore, the implemented saturation feature was tested in these mock-up projects as well showing a realistic developement of saturation and also covering cases with unrealistic developement of saturation.

8 Conclusion

In this thesis, the two inter-coder agreement metrics, Krippendorff's Alpha and Fleiss' Kappa have been implemented to QDAcity. They were embedded in a generic architecture, which allows the implementation of further inter-coder agreement metrics and supports the generation of generic reports to view the results of the different metrics in flexible tables. Furthermore, a new approach for calculation of theoretical saturation is presented in detail including the mathematical description of the formulas used for the calculation of saturation and the implementation details in QDAcity. The correctness of the implementations in this thesis was tested using unit tests. The user's benefit was verified by conducting manual tests using the QDAcity frontend using mock-up projects covering the QDAcity use cases and also containing chaotic data.

8.1 Future Work

This thesis can be used as a basis for further research or further implementations.

Regarding inter-coder agreement metrics, further metrics can be implemented to QDAcity, as the process of generating reports with these metrics supports any feasible inter-coder agreement metric as discussed in section 5.2. Having several metrics in QDAcity enables the possibility to run different metrics on the same data and contrast their benefits in the context of QDA.

The implemented approach for inter-coder agreement metrics also supports the use of different units of coding. However, only the coding unit *paragraph* was implemented and could be extended for example with the coding unit *sentences*.

Additionally, performance optimization on GAE brings up further questions like how to improve scalability by using different instance types in combination with different type of TaskQueues (see subsection 3.1.4) or the optimal granularity of tasks in the TaskQueues.

Glossary

JUnit Unit testing framework for the Java programming language. iv, 39, 40

MaxQDA Software to support Qualitative Data Analysis. 2

Acronyms

CSV Comma Separated Value. 16

GAE Google App Engine. 1, 5, 6, 7, 8, 14, 15, 34, 36, 38, 41

HTML Hypertext Markup Language. 6, 11

HTTP HyperText Transfer Protocol. 15, 33

HTTPS HyperText Transfer Protocol Secure. 6, 15

JDO Java Data Objects. 6

JSON JavaScript Object Notation. 6

MVC Model View Controller. 6

NLP Natural Language Processing. 13

NoSQL No Structured Query Language. 6

QDA Qualitative Data Analysis. 1, 3, 41

REST Representational State Transfer. 6

UML Unified Modeling Language. 9, 10

References

Artstein, R. & Poesio, M. (2008). Inter-coder agreement for computational linguistics. *Computational Linguistics*, *34*(4), 555–596.

Bowen, G. A. (2008). Naturalistic inquiry and the saturation concept: a research note. *Qualitative Research*, *8*(1), 137–152. doi:10.1177/1468794107085301. eprint: http://dx.doi.org/10.1177/1468794107085301

Bruce, R. F., Wiebe, J. et al. (1998). Word-sense distinguishability and inter-coder agreement. In *Emnlp* (pp. 53–60).

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, *20*(1), 37–46. doi:10.1177/001316446002000104. eprint: http://dx.doi.org/10.1177/001316446002000104

Corbin, J. & Strauss, A. (1990). Grounded theory research: procedures, canons and evaluative criteria. *Zeitschrift für Soziologie*, *19*(6), 418–427.

Erich Gamma, R. J., Richard Helm & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological bulletin*, *76*(5), 378–382.

Francis, J. J., Johnston, M., Robertson, C., Glidewell, L., Entwistle, V., Eccles, M. P., & Grimshaw, J. M. (2010). What is an adequate sample size? operationalising data saturation for theory-based interview studies. *Psychology and Health*, *25*(10), 1229–1245.

Fusch, P. I. & Ness, L. R. (2015). Are we there yet? data saturation in qualitative research. *The Qualitative Report*, *20*(9), 1408.

Glaser, B. G., Strauss, A. L., & Strutzel, E. (1968). The discovery of grounded theory; strategies for qualitative research. *Nursing research*, *17*(4), 364.

Google Inc. (2008). Google App Engine Blog: Introducing Google App Engine + our new blog. https://googleappengine.blogspot.de/2008/04/introducing-google-app-engine-our-new.html. [Online; accessed 13.06.2017].

Google Inc. (2017a). An Overview of App Engine — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/an-overview-of-app-engine#scaling_types_and_instance_classes. [Online; accessed 25.07.2017].

Google Inc. (2017b). appengine-web.xml Reference — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/config/appref. [Online; accessed 18.08.2017].

Google Inc. (2017c). Architecture: Web Application on Google App Engine — Architectures — Google Cloud Platform. https://cloud.google.com/solutions/architecture/webapp. [Online; accessed 13.06.2017].

Google Inc. (2017d). Cloud Datastore Overview — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/datastore/. [Online; accessed 11.06.2017].

Google Inc. (2017e). Java Datastore API — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/datastore/api-overview. [Online; accessed 18.06.2017].

Google Inc. (2017f). Memcache Overview — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/memcache/. [Online; accessed 13.06.2017].

Google Inc. (2017g). Push Queues in Java — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/taskqueue/push/. [Online; accessed 24.07.2017].

Google Inc. (2017h). Task Queue Overview — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/taskqueue/. [Online; accessed 11.06.2017].

Google Inc. (2017i). Using Pull Queues in Java — App Engine standard environment for Java — Google Cloud Platform. https://cloud.google.com/appengine/docs/standard/java/taskqueue/overview-pull. [Online; accessed 24.07.2017].

Google Inc. (2017j). Writing and Annotating the Code — Cloud Endpoints — Google Cloud Platform. https://cloud.google.com/endpoints/docs/frameworks/java/annotate-code. [Online; accessed 13.06.2017].

Guest, G., Bunce, A., & Johnson, L. (2006). How many interviews are enough? an experiment with data saturation and variability. *Field methods*, *18*(1), 59–82.

Joyce, M. (2013). Picking the best intercoder reliability statistic for your digital activism content analysis. In *Digital activism research project: investigating the global impact of comment forum speech as a mirror of mainstream discourse* (Vol. 243).

Kerr, C., Nixon, A., & Wild, D. (2010). Assessing and demonstrating data saturation in qualitative inquiry supporting patient-reported outcomes research. *Expert review of pharmacoeconomics & outcomes research*, *10*(3), 269–281.

Kolbe, R. H. & Burnett, M. S. (1991). Content-analysis research: an examination of applications with directives for improving research reliability and objectivity. *Journal of Consumer Research*, *18*(2), 243. doi:10.1086/209256.

eprint: /oup/backfile/content_public/journal/jcr/18/2/10.1086/209256/2/18-2-243.pdf

Krippendorff, K. (1970). Estimating the reliability, systematic error and random error of interval data. *Educational and Psychological Measurement*, *30*(1), 61–70.

Krippendorff, K. (2004). Reliability in content analysis. *Human communication research*, *30*(3), 411–433.

Krippendorff, K. (2007). Computing krippendorff's alpha reliability. *Departmental papers (ASC)*, 43.

Krippendorff, K. (2011). Agreement and information in the reliability of coding. *Communication Methods and Measures*, *5*(2), 93–112.

Krippendorff, K. (2012). *Content analysis: an introduction to its methodology*. Sage Publications.

Landis, J. R. & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, 159–174.

MacQueen, K. M., McLellan, E., Kay, K., & Milstein, B. (1998). Codebook development for team-based qualitative analysis. *CAM Journal*, *10*(2), 31–36.

Miles, M. B. & Huberman, A. M. (1994). *Qualitative data analysis: an expanded sourcebook*. Sage Publications.

O'Reilly, M. & Parker, N. (2013). 'unsatisfactory saturation': a critical exploration of the notion of saturated sample sizes in qualitative research. *Qualitative Research*, *13*(2), 190–197. doi:10.1177/1468794112446106. eprint: http://dx.doi.org/10.1177/1468794112446106

Powers, D. M. (2011). Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation.

Powers, D. M. (2012). The problem with kappa. In *Proceedings of the 13th conference of the european chapter of the association for computational linguistics* (pp. 345–355). Association for Computational Linguistics.

Ricardo Baeza-Yates, B. R.-N. (1999). *Modern information retrieval*. ACM Press books. ACM Press. Retrieved from https://books.google.de/books?id=usHuAAAAMAAJ

Saldaña, J. (2015). *The coding manual for qualitative researchers*. Sage.

Scott, W. A. (1955). Reliability of content analysis:the case of nominal scale coding. *Public Opinion Quarterly*, *19*(3), 321. doi:10.1086/266577. eprint: /oup/backfile/content_public/journal/poq/19/3/10.1086/266577/2/19-3-321.pdf

Sim, J. & Wright, C. C. (2005). The kappa statistic in reliability studies: use, interpretation, and sample size requirements. *Physical therapy*, *85*(3), 257–268.

Spencer, L., Ritchie, J., Lewis, J., & Dillon, L. (2003). Quality in qualitative evaluation: a framework for assessing research evidence.

Viera, A. J., Garrett, J. M. et al. (2005). Understanding interobserver agreement: the kappa statistic. *Fam Med*, *37*(5), 360–363.